SELint : An SEAndroid policy analysis tool

Reshetova, Elena

SCITEPRESS Science And Technology Publications
2017

Reshetova , E , Bonazzi , F & Asokan , N 2017 , SELint : An SEAndroid policy analysis tool . in ICISSP : Proceedings of the 3rd International Conference on Information Systems Security and Privacy . SCITEPRESS Science And Technology Publications , pp. 47-58 , International Conference on Information Systems Security and Privacy , Porto , Portugal , 19/02/2017 . https://doi.org/10.5220/0006126600470058

http://hdl.handle.net/10138/307775
https://doi.org/10.5220/0006126600470058

SELint: An SEAndroid Policy Analysis Tool

Elena Reshetova[1], Filippo Bonazzi[2] and N. Asokan[2,3]
[1]*Intel OTC, Helsinki, Finland*
[2]*Aalto University, Helsinki, Finland*
[3]*University of Helsinki, Helsinki, Finland*
elena.reshetova@intel.com, filippo.bonazzi@aalto.fi, asokan@acm.org

Keywords: Security, SEAndroid, SELinux, Android, Access Control, Policy Analysis.

Abstract: SEAndroid enforcement is now mandatory for Android devices. In order to provide the desired level of security for their products, Android OEMs need to be able to minimize their mistakes in writing SEAndroid policies. However, existing SEAndroid and SELinux tools are not very useful for this purpose. It has been shown that SEAndroid policies found in commercially available devices by multiple manufacturers contain mistakes and redundancies. In this paper we present a new tool, SELint, which aims to help OEMs to produce better SEAndroid policies. SELint is extensible and configurable to suit the needs of different OEMs. It is provided with a default configuration based on the AOSP SEAndroid policy, but can be customized by OEMs.

1 INTRODUCTION

During the past decade Android OS has become one of the most common mobile operating systems. However, at the same time we have seen a big increase in the number of malware and exploits available for it (Zhou and Jiang, 2012; Smalley and Craig, 2013). Many classical Android exploits, such as GingerBreak and Exploid, attempted to target system daemons that ran with elevated - often unlimited - privileges. A successful compromise of such daemons results in the compromise of the whole Android OS, and the attacker would be able to obtain permanent root privileges on the device. Initially Android relied only on its permission system, based on Linux Discretionary Access Control (DAC), to provide security boundaries. However, after it became evident that DAC cannot protect from such exploits, a new Mandatory Access Control (MAC) mechanism has been introduced. SEAndroid (Smalley and Craig, 2013) is an Android port of the well-established SELinux MAC mechanism (Smalley et al., 2001) with some Android-specific additions. Similarly to SELinux, SEAndroid enforces a system-wide policy. The default SEAndroid policy was created from scratch and is maintained as part of the Android Open Source Project (AOSP)[1]. Starting from the 5.0 Lollipop release, Android requires every process to run inside a confined SEAndroid domain with a proper set of access control rules defined. This has forced many Android Original Equipment Manufacturers (OEMs) to hastily define the set of access control domains and associated rules needed for their devices. Our recent study (Reshetova et al., 2016) showed that all OEMs modify the default SEAndroid policy provided by AOSP due to many customizations implemented in their Android devices. The difficulty of writing well-designed SELinux policies together with high time-to-market pressure can possibly lead to the introduction of mistakes and major vulnerabilities. The study classified common mistake patterns present in most OEM policies and concluded that new practical tools are needed in order to help OEMs avoid these mistakes. In this paper we make the following contributions:

- Design of **a new, extensible tool, SELint,** that aims to help Android OEMs to overcome common challenges when writing SEAndroid policies (Section 4). In contrast to existing SELinux and SEAndroid tools (described in Section 3), it can be used by a person without a deep understanding of SEAndroid, given the initial configuration by an expert. The community can write new analysis modules for SELint in the form of SELint plugins. This is especially important given that the SEAndroid policy format changes with every release, and new notions and mechanisms are introduced

Up-to-date version of this paper available at arxiv.org/abs/1608.02339

[1]source.android.com

Reshetova, E., Bonazzi, F. and Asokan, N.
SELint: An SEAndroid Policy Analysis Tool.
DOI: 10.5220/0006126600470058
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 47-58
ISBN: 978-989-758-209-7

by Google.

- **An initial configuration** for SELint, based on the AOSP SEAndroid policy, which **was found to be useful** by the SEAndroid community in our evaluation of SELint (Section 5.1).
- A full **implementation** of SELint that that **fits OEM policy development workflows**, providing **reasonable performance** and allowing **easy customization** by OEMs (Section 5).

2 BACKGROUND

2.1 SELinux and SEAndroid

SELinux (Smalley et al., 2001) was the first mainline MAC mechanism available for Linux-based distributions. Compared to other mainline MAC mechanisms present today in the Linux kernel, it is considered to be the most fine-grained and the most difficult to understand and manage due to the lack of a minimal policy (like in Smack (Schaufler, 2008)) or a learning mode (like in AppArmor (Bauer, 2006)). Despite this, it is enabled by default in Red Hat Enterprise Linux (RHEL) and Fedora with pre-defined security policies.

The core part of SELinux is its Domain/Type Enforcement (Badger et al., 1995) mechanism, which assigns a `domain` to each subject, and a `type` to each object in the system. A subject running in `domain` can only access an object belonging to `type` if there is an `allow` rule in the policy of the following form:

```
allow domain type :  class permissions
```

where `class` represents the nature of an object such as file, socket or property, and `permissions` represent the kinds of operations being permitted on this object, like `read`, `write`, `bind` etc.

The **SEAndroid** (Smalley and Craig, 2013) MAC mechanism is mostly based on SELinux code with some additions to support Android-specific mechanisms, such as the Binder Inter Process Communication (IPC) framework. However, SEAndroid's policy is fully written from scratch and is very different from SELinux's reference policy. AOSP predefines a set of application domains, like `system_app`, `platform_app` and `untrusted_app`; applications are assigned to these domains based on the signature of the Android application package file (`.apk`). Other services and processes are assigned to their respective domains based on filesystem labeling or direct domain declaration in the service definition in the `init.rc` file. One notable feature of the SEAndroid policy is active usage of predefined M4 macros that make the policy more readable and compact. For example, the `global_macros` file defines a number of M4 macros that denote sets of typical permissions needed for common classes, such as `r_file_perms` or `w_dir_perms`. Another example is the `te_macros` file, that provides a number of M4 macros used to combine sets of rules commonly used together.

2.2 SEAndroid OEM Challenges

The SEAndroid reference policy only covers default AOSP services and applications. Therefore, highly customized OEM Android devices require extensive policy additions.

Our already mentioned study of different OEM SEAndroid policies for Android 5.0 Lollipop (Reshetova et al., 2016) showed that most OEMs made a significant number of additions to the default AOSP reference policy. The biggest changes are the additions of new types and domains, as well as new `allow` rules. The study also identified a number of common patterns that most OEM policies seem to follow:

- **Overuse of Default Types.** SEAndroid declares a set of default types that are assigned to different objects unless a dedicated type is defined in the policy. Most OEMs leave many such types in their policies, which indicates a use of automatic policy creation tools such as `audit2allow` (SELinux, 2014).
- **Overuse of Predefined Domains.** OEMs do not typically define dedicated domains for their system applications, but tend to assign these applications to predefined `platform_app` or `system_app` domains. This creates over-permissive application domains and violates the principle of least privilege.
- **Forgotten or Seemingly Useless Rules.** OEM policies have many rules that seem to have no effect. This might be due to an automatic rule generation or a failure to clean up unnecessary rules that were no longer required.
- **Potentially Dangerous Rules.** A number of potentially dangerous rules can be seem in some OEM policies, including granting additional permissions to `untrusted_app` domain. This might be due to lack of time to adjust their service or application implementation to minimize security risks or due to inability to identify some rules as being dangerous.

3 RELATED WORK

Since SELinux existed on its own long before SEAndroid, most of the available tools are designed to handle and analyze SELinux policies. They can be used for SEAndroid but they don't take specific aspects of SEAndroid policies into account. This makes it challenging for OEMs to use existing tools to detect the problems outlined in Section 2.2. For example, in order to determine if the policy contains potentially dangerous rules, it is very important to understand the semantics of SEAndroid types and policy structure - an ability which all existing SELinux tools lack. Moreover, even the small group of SEAndroid tools described in Section 3.2 does not address the challenges described in Section 2.2.

3.1 SELinux Tools

SETools (Tresys, 2016) is the official collection of tools for handling SELinux policies in text and binary format. Some of its tools, like `apol`, are suitable for formal policy analysis, for example for flow-control analysis. Others allow policy queries and policy parsing and as such it can be used on both SELinux and SEAndroid. An important part of SETools is a policy representation library which is used in both SEAL and SELint.

Formal methods have been applied to SELinux policy analysis. Gokyo (Jaeger et al., 2003) is a tool designed to find and resolve conflicting policy specifications. Guttman *et al.* (Guttman et al., 2005) applies information flow analysis to SELinux policies. The HRU security model (Harrison et al., 1976) has been used to analyze SELinux policies (Amthor et al., 2011). Hurd *et al.* (Hurd et al., 2009) applied Domain Specific Languages (DSL) (Fowler, 2010) in order to develop and verify the SELinux policy. The resulting tool, `shrimp`, can be used to analyze and find errors in the SELinux Reference Policy. Information visualization techniques have been applied to SELinux policy analysis in (Clemente et al., 2012), also in combination with clustering of policy elements (Marouf and Shehab, 2011). These analysis methods are largely academic, and no practical tools based on them are used by the SELinux community.

Polgen (Sniffen et al., 2006) is a tool for semi-automated SELinux policy generation based on system call tracing. Unfortunately it appears to be no longer in active development. SELinux also provides a set of userspace tools (SELinux, 2014) that can be used on both SELinux and SEAndroid. One of these tools, `audit2allow`, is widely used by Android OEMs to automatically generate and expand SEAndroid policies. The tool works by converting denial audit messages into rules based on a given binary policy. These rules, however, are not necessarily correct, complete or secure, since they entirely depend on code paths taken during execution and require a good understanding of the software components involved, as well as on the correct labeling of subjects and objects in the system. Furthermore, automatically-generated rules fail to use high-level SEAndroid policy features such as attributes and M4 macros: this results in comparatively less readable policies.

3.2 SEAndroid Tools

Our aforementioned study (Reshetova et al., 2016) presented SEAL, an SEAndroid live device analysis tool. SEAL works with a real or emulated Android device over the Android Debug Bridge (ADB); it can perform different queries that take into account not only the binary SEAndroid policy loaded on the device, but also the actual device state, i.e. running processes and filesystem objects. The EASEAndroid policy refinement method is based on audit log analysis with machine learning (Wang et al., 2015). This approach is completely different from what we propose, since it relies on significant volumes of data to classify rules. Unfortunately, it is very hard to obtain this volume of data, since it would require collecting log files from millions of Android devices with possible privacy implications. The most recent SEAndroid policy analysis and refinement tool is SEAndroid Policy Knowledge Engine (SPOKE) (Wang, 2016). It automatically extracts domain knowledge about the Android system from application functional tests, and applies this knowledge to analyze and highlight potentially over-permissive policy rules. SPOKE can be used to identify new heuristics that can be implemented as new SELint plugins. The downside of SPOKE is reliance on application functional tests, which are often incomplete, and the fact that it cannot be easily integrated into the standard development workflow.

4 SELint

4.1 Requirements

We identify the following generic requirements that a tool like SELint must fulfill.

R 1. Source Policy-based. The existing tool landscape presented in Section 3 does not feature any tool able to perform semantic analysis on source SEAndroid policies. Since Android OEMs work on source

SEAndroid policies as part of their Android trees, the tool needs to work with source SEAndroid policies.

R 2. Configurable by Experts, Usable by All. Existing tools require extensive domain knowledge to be used. Since building such a knowledge takes considerable time, it might be challenging for OEMs to have all of their development team trained appropriately. We intended for our tool to fit into an Android OEM policy development workflow, where many developers, overseen by one or a few experienced SEAndroid analysts, contribute small changes to the policy. Therefore, it must be possible for an experienced analyst to configure the tool ahead of time, and provide a ready-to-run tool to regular developers, who can simply run the tool on their policy modifications and verify that no issues are highlighted.

R 3. Reasonable Performance. Since we are targeting inclusion into an Android OEM workflow, the tool must have reasonable time and memory performance; this is necessary for the tool to be used as part of the build toolchain, or even more appropriately when committing changes using the OEM's version control software (VCS).

R 4. Easy to Configure and Extend. Finally, targeting the wide community of Android OEMs makes it impossible to know in advance all possible use cases and requirements, present and future. It is our objective to allow analysts to implement their own analysis functionality and embed their domain knowledge into the tool. For this reason, the tool must be easily configurable and extensible by the community.

4.2 General Architecture and Implementation

To meet Requirement 4 stated in section 4.1, we designed SELint following a plugin architecture. The goal of such an architecture is to support custom third-party analysis plugins that any community member can create. The core part of SELint is responsible for processing the source SEAndroid policy. The *SELint core* takes care of handling user input, such as command line options and configuration files. After the source policy has been parsed, its representation is given to the *SELint plugins* which perform the actual analysis. We have developed an initial set of plugins, which provide generally useful functionality; interested Android OEMs can develop more plugins to implement their own analysis requirements.

The overall architecture is shown in Figure 1; the existing plugins are individually described in the following sections. The implementation of SELint and the existing plugins are released under the Apache License 2.0, which allows the community to freely use and modify the software. The `policysource` library is released under the GNU Lesser General Public License v2.1.

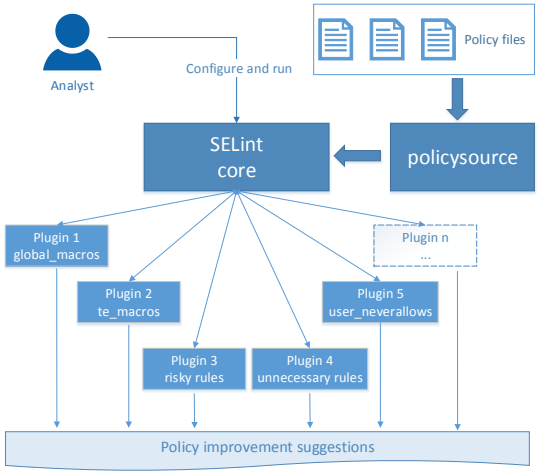

Figure 1: The architecture of SELint.

The SELint executable and all plugins have an associated configuration file. This allows policy experts to adapt each plugin to the semantics of their own policies, for example to define OEM-specific policy types. This way, SELint can be run with different preset "*profiles*" specifying different options, policy configurations and requested analysis functionalities. The following sections describe each existing SELint plugin in detail.

4.3 Plugin 1: Simple Macros

Goal As mentioned in Section 2.1, using M4 macros

```
r_file_perms → { getattr open read ioctl lock }
```

Figure 2: A `global_macros` definition and expansion.

where applicable is a non-functional requirement of SEAndroid policy development: while not affecting policy behavior, their use makes for a more compact and readable policy. The first type of M4 macros extensively used in SEAndroid policies is a simple text replacement macro, without arguments, that is used to represent sets of related permissions. Such macros are defined in the `global_macros` file in the SEAndroid policy source files. An example of such macro is shown in Figure 2. The Simple macro plugin scans the policy for rules granting sets of individual permissions which could be represented in a more compact way by using an existing `global_macros` macro; it then suggests replacing the individual permissions with an usage of said macro.

Implementation. The plugin looks for rules which specify individual permissions whose combination

is equivalent to the expansion of a `global_macros` macro. It then suggests rewriting said rules, replacing the individual permissions with the unexpanded macro. An example is shown in Figure 3. For this particular case, the plugin suggest replacing a set of permissions {`gettattr open read search ioctl`} with a macro `r_dir_perms`. Permissions not contained in the macro (in this case `create`) are still specified individually in the final rule. The plugin can suggest both full matches (for rules that grant 100% of the permissions contained in a macro) and partial matches above a threshold (for rules that grant at least X% of the permissions contained in a macro). This threshold is a user-defined parameter, specified in the plugin configuration file; we assigned it a default value of 0.8 (80%).

Rule:
```
allow logd rootfs:dir
 {getattr create open read search ioctl};
```

Macro:
`r_dir_perms` → {`open getattr read search ioctl`}

Suggestion:
```
allow logd rootfs:dir {r_dir_perms create};
```

Figure 3: An example usage of the Simple macro plugin.

Limitations. The plugin only deals with simple, static macros without arguments. Dynamic macros such as those defined in the `te_macros` file are handled by the dedicated plugin described in the next section.

4.4 Plugin 2: Parametrized Macros

Goal.

```
        'file_type_trans($1, $2, $3)'
                        ↓
'allow $1 $2:dir ra_dir_perms;
 allow $1 $3:dir create_dir_perms;
 allow $1 $3:notdevfile_class_set
                              create_file_perms;'
```

Figure 4: A `te_macros` macro definition and expansion with arguments.

Another commonly used set of M4 macros includes more complex, dynamic M4 macros with multiple arguments. Such macros are mainly used to group rules which are commonly used together; their expansion can in turn contain other macros. In SEAndroid policies, such macros are defined in the `te_macros` file. An example is shown in Figure 4. Similarly to the previous, this plugin detects existing macro definitions, and suggests new usages.

Implementation. The plugin looks for sets of individually specified rules whose combination is equivalent to the expansion of a `te_macros` macro with some set of arguments. It then suggests substituting said rules with a usage of the unexpanded macro with the proper arguments. An example is shown in Figure 5: the plugin finds the existing macro which expands into the given set of rules - in this case, `unix_socket_connect`. It then extracts the arguments from the rules: $1 is "a", $2 is "b" and $3 is "c". The result is a suggestion for substituting the two rules with the macro usage. The plugin can suggest both full matches (for sets of rules that match 100% of the rules contained in a macro expansion) and partial matches above a user-defined threshold. Its default value is 0.8 (80%).

Rules:
```
allow a b_socket:sock_file write;
allow a c:unix_stream_socket connectto;
```

Macro:
```
unix_socket_connect($1, $2, $3)
                ↓
allow $1 $2_socket:sock_file write;
allow $1 $3:unix_stream_socket connectto;
```

Suggestion:
```
unix_socket_connect(a, b, c)
```

Figure 5: An example usage of the Parametrized macro plugin.

Limitations. The problem of detecting sets of rules that match possible macro expansions can be transformed into a variant of the knapsack problem (Kellerer et al., 2004), namely a multidimensional knapsack problem. In our case, the knapsack capacity is the number of arguments a macro can have, and the knapsack items are the possible values of these arguments; the knapsack is multidimensional because filling an argument does not affect the available capacity for the others. Instead of finding the single most profitable combination of argument-values, our objective is to find all the combinations of argument-values which, used as arguments in as many macro expansions, produce sets of rules entirely or partially (above a threshold) contained in the policy. The problem can be formalized as:

For each macro m, find all combinations of values for arguments $1, $2 and $3 such that y is above an user-given threshold t. y is computed as: $y = score(m(i,j,k))$, subject to $i \in N_i, j \in N_j|_i, k \in N_k|_{ij}$, where N_i is the set of possible values of $1, $N_j|_i$ is the set of possible values of $2 given i as $1, and $N_k|_{ij}$ is the set of possible values of $3 given i as $1 and j as $2. $score(m(i,j,k))$ is the score of the macro expanded with the arguments i, j and k: the score of a macro expansion is given by the number of its

rules actually found in the policy divided by its overall number of rules.

The multidimensional knapsack optimization problem is known to be NP-hard (Magazine and Chern, 1984), and it has various approximate solutions (Chu and Beasley, 1998; Hanafi and Freville, 1998). In our case, the problem quantities are the number of arguments a macro can have (existing macros have 1-3), the number of rules a macro expansion can produce (existing macros have 1-7), and the number of values a macro argument can have (in principle infinite, in practice dependent on the policy, usually in the thousands). In practice, the number of rules (#2) tends to increase linearly with the number of arguments (#1). This is due to the fact that macros with more arguments can define more complex behavior, which tends to be described in more rules.

As a first implementation, we realized a simple solution based on exploration of the solution space: we try to aggregate all the policy rules into sets corresponding to macro expansions. The problem quantities described above result in a significant time expenditure required to explore the whole solution space: therefore, as we discuss in Section 5.2, this plugin takes considerably more time than all others.

4.5 Plugin 3: Risky Rules

Experts analyzing OEM modifications to SEAndroid policies often use certain heuristics. The analysis usually starts from the list of AOSP SEAndroid domains and types that are more likely to cause potential vulnerabilities in OEM policies. The most common are:

- **Untrusted Domains.** Some domains are intended to run potentially malicious code, such as `untrusted_app`, and therefore their privileges are designed to be minimal. Any additional `allow` rules created by OEMs for such domains are suspicious and need to be analyzed.
- **Trusted Computing Base (TCB) Domains and Types.** The AOSP policy has several core domains and types, which form its TCB. The processes that run in these domains are provided by AOSP, and so are the minimal required policy rules. Sometimes, OEMs have to create additional rules for some of these domains: however, since doing so increases the chance of compromising the TCB, such rules need thoughtful inspection.
- **Security-related Domains and Types.** Special attention must be paid to AOSP domains and types directly related to system security, such as the `tee` domain or the `proc_security` type. Mistakes in additional `allow` rules for these domains and types can lead to a direct loss of system security.

An analyst usually checks an OEM policy for additional rules where the above domains or types are present, and then manually inspects each rule analysing its domain, type and permissions to determine if the rule is actually risky. This process is tedious, and most of the time is spent just finding the rules which need special attention. To help analysts find these rules quickly, we developed the `risky_rules` plugin, which processes each rule and assigns it a score based on one of two criteria.

The first scoring criterion is based on *risk*. We define the **risk level** for rule components as the level of potential damage to the system caused by misuse of the component: security-sensitive components will have high risk scores, while generic components will have lower risk scores. Untrusted domains will have a high *risk* score as well, because we want to select any additional rules over such domains for manual inspection. Component risk level in turn determines the risk level of a rule, which is obtained by combining the risk levels of its components. The risk level of a rule is then defined as the level of potential damage to the system allowed by the rule. The risk score helps analysts to quickly obtain a prioritized list of policy rules which need manual inspection; this is especially useful when analysts have strict time constraints, and only have time to examine a limited number of rules. The *risk* scoring system is described in Section 4.5.1.

The second scoring criterion is based on *trust*. We define the **trust level** for rule components as a measure of closeness to the core of the system: key system components will have a high *trust* level, while user applications will have a low *trust* level. This in turn allows us to detect rules which cross trust boundaries, *e.g.* comprising a *high* component and a *low* component or vice versa. This scoring system is useful for an analyst as well, because it can quickly identify additional OEM rules which breach trust boundaries and select them for manual inspection. The *trust* scoring system is described in Section 4.5.2.

The desired scoring system can be specified in the plugin configuration file. We have provided an initial `risky_rules` plugin configuration based on our knowledge and experience with the AOSP policy. While our classification might be considered subjective, feedback discussed in Section 5.1 indicates that SEAndroid policy writers agree with our approach.

4.5.1 Measuring Risk

Goal. As mentioned above, rules in a policy can have different risk levels, depending on the types they deal

with and the permissions they grant. The *risk* scoring system of the `risky_rules` plugin assigns a score to every rule in the policy, prioritizing potentially riskier rules by assigning them higher scores.

Implementation. The *risk* scoring system computes the overall score for a rule by evaluating its domain, type, and permissions or capabilities. The plugin configuration file defines partial *risk* scores for various rule elements. Relevant AOSP domains and types are grouped by risk level into "*bins*", which are assigned a partial *risk* score with a maximum of 30. When computing the score for a rule, the partial scores of its domain and type are added. We treat domains and types equally, because both the running process and data of a program might be equally important in evaluating how risky a rule is. For example, a process running in a security sensitive domain (*e.g.* `keystore`) should not accept any command from other processes running in unauthorized domains, because they might induce malicious changes in its execution flow. Similarly, other unauthorized processes should not be able to modify the configuration data of a security sensitive process (*e.g.* data labeled as `keystore_data`), for similar reasons. The initial set of bins and their default scores are depicted in Table 1.

Table 1: `risky_rules` plugin default bins and partial *risk* scores.

Bin name	Example types	Risk
`user_app`	`untrusted_app`	30
`security_sensitive`	`tee, keystore, security_file`	30
`core_domains`	`vold, netd, rild`	15
`default_types`	`device, unlabeled, system_file`	30
`sensitive`	`graphic_device`	20

The `user_app`, `core_domains` and `security_sensitive` bins match groups defined earlier in this section. `user_app` and `security_sensitive` have the maximum score of 30, while the score for `core_domains` is 15 due to less overall risk to the system. The `default_types` bin has a maximum score of 30, because it contains types that should not normally be used by OEMs and therefore likely indicate a mistake in a rule.

When computing the overall *risk* score for a rule, in addition to evaluating a rule's domain and type elements, the *risk* scoring system must also take its permissions and capabilities into account. In SEAndroid, permissions are meaningless in isolation, and only meaningful to determine risk when combined with the domain to which they are granted and the type over which they are granted: for this reason, we combine these when computing the *risk* score for a rule. We do this by assigning permissions a multiplicative coefficient instead of an additive partial score; the sum of domain and type score for a rule is multiplied by this coefficient. Commonly used permissions are categorized by level of risk into three groups, `perms_high`, `perms_med` and `perms_low`: each group is assigned a coefficient based on the sensitivity of its permissions, with a maximum of 1. The sum of domain and type score is multiplied by the coefficient of the highest set which contains permissions granted by the rule; this is done because we are interested in determining the upper bound of risk for a rule. Table 2 shows the groups, permissions and default values of coefficients.

Table 2: `risky_rules` plugin default permission sets and coefficients.

Set name	Example permissions	Coefficient
`perms_high`	`ioctl, write, execute`	1
`perms_med`	`read, use, fork`	0.9
`perms_low`	`search, getattr, lock`	0.5

Capabilities are treated differently from permissions. In SEAndroid, capabilities are granted by a domain to itself, and - unlike permissions - are meaningful on their own: they have the same effect on the system regardless of the domain they are granted to. For example, the following rule grants the `vold` daemon the `CAP_CHROOT` capability, which allows it to perform the `chroot` system call:

```
allow vold self:capability sys_chroot;
```

We do not divide capabilities into separate groups: this is due to the fact that, in Linux, capabilities are commonly believed to be very hard to categorize as more or less dangerous, because of the consequences they can have on the system[2]. Since in SEAndroid capabilities are granted by a domain to itself, the target type in such a rule does not convey any additional information: therefore, we use a special scoring formula for rules granting capabilities. Capabilities are handled as types, and any capability is assigned the maximum score for a type (30): this score is added to the domain score to obtain the rule score.

The *risk* scoring system scores rules by their potential level of risk between 0 and 1, with maximum risk given a score of 1. As discussed above, risk scores are assigned to rules depending on the type of rule: the precise formulas are presented in Figure 6.

An example is shown in Figure 7. The first rule contains `untrusted_app` and `security_file`, which are both high-risk types (`user_app` and `security_sensitive` respectively); however, the rule only grants the `getattr` and `search` permissions, which are two low-risk permissions. Thus, the

[2]forums.grsecurity.net/viewtopic.php?f=7&t=2522

Allow rules granting permissions :

$$\text{score}_{risk}(rule) = \frac{\text{score}_{risk}(domain)+\text{score}_{risk}(type)}{M} \cdot C$$

$$C = \max_{0 \le i < nperms}(\text{coefficient}_{risk}(perm_i))$$

Allow rules granting capabilities:

$$\text{score}_{risk}(rule) = \frac{\text{score}_{risk}(domain)+\text{score}_{risk}(capabilities)}{M}$$

Type_transition rules:

$$\text{score}_{risk}(rule) = \frac{\text{score}_{risk}(domain)+\text{score}_{risk}(type)}{M}$$

M is the maximum value of the numerator (60), used to normalize the score between 0 and 1.

Figure 6: The *risk* scoring formulas for the `risky_rules` plugin.

rule has a medium *risk* score that in this case equals to 0.5. The second rule contains `untrusted_app` and `system_file`, which are both high-risk types (`user_app` and `default_types` respectively); furthermore, the rule grants the `execute` permission, which is a high-risk permission. Thus, the rule has a high *risk* score that in this case equals to 1.

```
0.50: .../domain.te:154: allow untrusted_app
      security_file:dir { getattr search };
1.00: .../domain.te:104:
   allow untrusted_app system_file:file execute;
```

Figure 7: An example of the `risky_rules` plugin with the *risk* scoring system.

4.5.2 Measuring Trust

Goal Rules in a policy can contain domains and types with different *trust* levels. Analysts usually inspect a policy by manually looking for rules which cross *trust* boundaries and making sure they are justified: this process is time-consuming and can be error prone. The *trust* scoring system of the `risky_rules` plugin automates this search: it assigns a score to every rule in the policy, prioritizing rules which cross *trust* boundaries by assigning them higher scores.

Implementation. The *trust* scoring system combines the partial scores of domain and type in a rule to assign it an overall score. The plugin configuration file defines partial *trust* scores for various rule elements. AOSP domains and types are grouped into "*bins*", which are assigned a *trust* score with a maximum of 30. When computing the score for a rule, the partial scores of its domain and type are added. The initial bins with their default scores are depicted in Table 3.

For example, the `user_app` bin contains types assigned to generic user applications, such as

Table 3: `risky_rules` plugin default bins and partial *trust* scores.

Bin name	Example types	Trust
`user_app`	`untrusted_app`	0
`security_sensitive`	`tee, keystore, security_file`	30
`core_domains`	`vold, netd, rild`	20
`default_types`	`device, unlabeled, system_file`	5
`sensitive`	`graphic_device`	10

`untrusted_app`; since user applications are not trusted, the *trust* score for this bin is minimum (0). The `security_sensitive` bin contains types assigned to data or components that have direct security impact, such as `tee`, `keystore`, `proc_security` *etc*. These components and their data are also highly trusted, since they form the TCB of the system, and therefore their *trust* score is maximum (30). The *trust* scoring system scores rules by the level of trust of their domain and type, regardless of the type of rule. Permissions and capabilities are ignored when computing the *trust* score for a rule. The level of trust can be *high* or *low*, giving place to 4 different scoring criteria: *trust_hl*, where the rule features a *high* domain and a *low* type, *trust_lh*, where the domain is *low* and the type is *high*, *trust_hh*, where both are *high*, and *trust_ll*, where both are *low*. The various *trust* criteria score rules between 0 and 1, where a score of 1 indicates that a rule is closest to the specified criterion. A high rule score is obtained naturally when looking for *high* components: to obtain a high rule score when looking for *low* components, the component partial score is subtracted from the maximum partial score before normalizing. Trust scores are assigned to rules using the formulas presented in Figure 8.

Trust_ll:

$$\text{score}_{trust}(rule) = \frac{(\frac{M}{2}-\text{score}_{trust}(domain))+(\frac{M}{2}-\text{score}_{trust}(type))}{M}$$

Trust_lh:

$$\text{score}_{trust}(rule) = \frac{(\frac{M}{2}-\text{score}_{trust}(domain))+(\text{score}_{trust}(type))}{M}$$

Trust_hl:

$$\text{score}_{trust}(rule) = \frac{(\text{score}_{trust}(domain))+(\frac{M}{2}-\text{score}_{trust}(type))}{M}$$

Trust_hh:

$$\text{score}_{trust}(rule) = \frac{\text{score}_{trust}(domain)+\text{score}_{trust}(type)}{M}$$

M is the maximum value of the numerator (60), used to normalize the score between 0 and 1.

Figure 8: The *trust* scoring formulas for the `risky_rules` plugin.

An example of one of the *trust* scoring systems (*trust_lh*) is shown in Figure 9. The first rule contains

untrusted_app, which is a low-trust domain, and system_file, which is a low-trust domain. The scoring criterion assigns the maximum score to rules with a *low* domain and a *high* type: therefore, the rule has a medium *trust_lh* score, which in this case is 0.58. The second rule contains untrusted_app, which is a low-trust domain, and security_file, which is a high-trust type. According to the selected scoring criterion, the rule has the maximum *trust_lh* score of 1.

```
0.58: .../domain.te:104:
    allow untrusted_app system_file:file execute;
1.00: .../domain.te:154: allow untrusted_app
        security_file:dir { getattr search };
```

Figure 9: An example of the risky_rules plugin with the *trust_lh* scoring system.

4.5.3 Limitations

Both scoring systems, *risk* and *trust*, assign a score to a rule by computing a formula over the partial scores of various rule elements. These partial scores must be defined by an analyst in the plugin configuration file, and simply reflect what an analyst is most interested in. Only the analyst who defined an element in the policy has the relevant knowledge to assign it a *risk* or *trust* score. A high rule score does not mean that a rule is dangerous, and a low score does not mean that a rule is safe: a high score represents a rule which the analyst deems more interesting, and vice versa.

4.6 Plugin 4: Unnecessary Rules

Goal. Some rules are effective only when used in combination. For example, a type_transition rule is useless without the related allow rules actually enabling the requested access. Similarly, some permissions are meaningful only when granted in combination. For example, an allow rule which grants read on a file type, without granting open on the same type or use on the related file descriptor type, will not actually allow the file to be read. Another example is debug rules, which are effective only when used for an OEM internal engineering build, and should not be present in the derived user build which is actually shipped. An analyst may want to check that all such rules are correctly wrapped inside debug M4 macros, which prevent them from appearing in the final user build. The unnecessary_rules plugin searches the policy for rules which are ineffective or unnecessary, as in the examples above. It also looks for debug rules mistakenly visible in the user policy.

Implementation. The plugin provides 3 features: detection of ineffective rule combinations, detection of debug rules, and detection of ineffective permissions.

Tuple:
```
type_transition $ARG0 $ARG1:file $ARG2;
allow $ARG0 $ARG1:dir { search write };
allow $ARG0 $ARG2:file { create write };
```
If found:
```
    type_transition a b:file c;
```
Look for:
```
    allow a b:dir { search write };
    allow a c:file { create write };
```

Figure 10: An example of the "ineffective rule combinations" functionality of the unnecessary_rules plugin.

Ineffective rule combinations: The plugin detects missing rules from an ordered tuple of rules. Tuples can be specified by an analyst in the plugin configuration file, and can contain placeholder arguments. This functionality looks for rules matching the first rule in a tuple, and verifies that all other rules in the tuple are present in the policy. An example is shown in Figure 10. The tuple contains three rules with placeholder arguments. If a rule is found matching the first rule in the tuple, the arguments are extracted and substituted in the remaining rules; each of these rules must then be found in the policy.

Debug Rules: The plugin detects rules containing debug types as either the domain or the type. Debug types can be specified by an analyst in the plugin configuration file.

Ineffective Permissions: The plugin detects rules which grant some particular permission on a type, but do not grant some other particular permission on that type or some additional permissions on some other (related) type. All three sets of permissions can be specified by an analyst in the configuration file. An example is shown in Figure 11. If any permissions from the first set are granted on a file, then either all the permissions in the second set must be granted on the file, or the permissions in the third set must be granted on the file descriptor. The first rule grants read and write from the first set, and does not grant open from the second set; however, the second rule grants use on the file descriptor. The constraint is therefore satisfied.

Limitations. The plugin allows an analyst to express very fine-grained information: this results in a some-

If found:
```
    file { write read append ioctl}
```
Look for either:
```
    file { open }
```
or:
```
    fd {use}
```
Rules:
```
    allow a b:file { read write };
    allow a b:fd use;
```

Figure 11: An example of the "ineffective permissions" functionality of the unnecessary_rules plugin.

what complex configuration file.

4.7 Plugin 5: User `neverallows`

Goal. `neverallow` rules can be used to specify permissions never to be granted in the policy. For example, Google uses `neverallow` rules extensively to prevent OEMs from circumventing core security structures of the policy. However, `neverallow` rules are only enforced at compile time in the normal SEAndroid policy development workflow: this means that a policy change may be committed into an OEM's VCS, only to later find out that it infringes one or more `neverallow` rules and therefore breaks the compilation. The `user_neverallows plugin` allows an analyst to define an additional set of `neverallow` rules, and be able to check at any time if they are respected by the policy. This can be very useful for OEM policy maintainers who would like to immediately make sure that developers contributing small policy changes do not introduce any undesired rules. The plugin enforces a list of custom user-defined `neverallow` rules on a policy, reporting any infringing rule.

Implementation. The plugin checks each rule in the policy which matches any user-specified `neverallow`, and verifies that it does not grant any permission explictly forbidden in the `neverallow`. Custom `neverallow` rules can be defined by the analyst in the plugin configuration file, in the same syntax as they would be written in the policy.

Limitations. The `user_neverallows` plugin processes each user-provided `neverallow` rule individually: therefore, it works best with small numbers of rules (tens of thousands).

5 EVALUATION

In order to show that SELint fulfills the requirements stated in Section 4.1, we solicited feedback from SEAndroid experts about their experience with SELint, as well as measured the tool's performance.

5.1 Expert Survey

Following Requirement 2, SELint is designed to be configured by an SEAndroid expert before regular developers can use it in their work flow. SEAndroid experts are, therefore, the main target audience of SELint. Developers are just expected to run SELint and verify that it doesn't produce new warnings on their policy modifications. Thus, in order to evaluate the usability and usefulness of SELint, we need to collect feedback from SEAndroid experts.

Materials. In order to collect expert feedback about SELint, we prepared an evaluation questionnaire[3]. SELint itself was available for download via our public Github repository[4].

Procedure. When collecting feedback on SELint, we wanted to focus on people that already have strong prior experience with SEAndroid policies. This choice is based on the fact that these experts are able to evaluate not just the tool itself, but also the default configuration we provide for its plugins. In order to obtain such feedback, we announced the SELint tool on the SEAndroid public mailing list[5]. This mailing list is a common forum where discussions among SEAndroid experts take place. We asked people to fill in the questionnaire after trying to use the tool on their Android tree.

Participants. Three experts from three different companies evaluated SELint. Each had more than 2 years of experience with SEAndroid policies.

Results. All respondents ranked SELint as easy to use, and its results as easy to interpret. They also agreed that functionality offered by SELint is not currently provided by any existing tools; they ranked SELint as being "*valuable*" for them for their current work on SEAndroid. Our free-form questions on the overall SELint experience gathered answers such as:

> "*I was able to use the tool to find things I wanted to fix with respect to over-privileged domains and useless rules.*"

> "*I think this just adds to the list of useful tools in policy development. The output is more user friendly than sepolicy-analyze and hopefully would appeal to those who only write policy infrequently - such as most OEMs.*"

Out of all the default plugins we provided with SELint, the `risky_rules` plugin caught the most attention and received the most positive feedback. This is as expected, given that this is the plugin that helps the most to directly evaluate the security of a SEAndroid policy. Plugins dealing with M4 macros were also found to be useful, with respondents reporting that they actually adopted most or all suggestions for `global_macros` or `te_macros` in their SEAndroid policy. The `neverallow_rules` plugin got an expected answer to the question "*Do you plan to use the neverallow_rules plugin?*":

> "*Yes, to add rules I don't want in the policy, but where I don't want to add an actual neverallow. Neverallows end up in CTS, so you don't want to*

[3] goo.gl/forms/j9oUBL2wnEjOvpLs2

[4] github.com/seandroid-analytics/selint

[5] seandroid-list@tycho.nsa.gov

use them too much. As for OEM policy additions, sometimes neverallows are too strict and we just want to see what the linter picks up."

This is exactly the usage we envisioned for it: an ability for OEMs to enforce custom `neverallow` rules without them being checked by Android Compatibility Test Suite (CTS). Respondents also had some good points for future enhancements, such as implementing an easier setup wizard and automatically prompting to input the scores for types or permissions which do not have one in the `risky_rules` plugin.

Limitations. In order to perform a better evaluation of SELint, we need a more extensive study with many more OEM developers who need to modify SEAndroid policies. However, this is difficult to achieve because of the following reasons. In order to try SELint, participants need to have their own custom Android tree and their own custom SEAndroid policies, since the tool targets OEM SEAndroid policy writers; this naturally limits the number of participants. In addition, people that actually have their own custom policy are usually engineers working for OEMs. They might not want to take part in our study because of corporate confidentiality concerns. Another difficulty is in setting up SELint, as one of our respondents noted. This is due to the fact that SELint relies on the policy representation library from SETools (Tresys, 2016) to perform policy parsing, and older versions of this library do not support some new SEAndroid policy elements, such as `xperms`. This, together with some compatibility issues between SEAndroid policy versions and SETools, made it harder for some users to setup the tool initially.

Despite these limitations, we believe the user feedback we received confirms that our goals and assumptions for SELint and the default configurations of its plugins are correct. In addition, this feedback gives us directions for future work discussed in Section 6. We also hope that we will receive more user feedback on our tool with time.

5.2 Performance Evaluation

Table 4: Performance measurements for SELint on Intel Android tree with 99532 expanded rules.

Component	Avg time (s)	Avg mem (MB)
SELint core	0.40 ± 0.01	99.53 ± 0.06
user_neverallows	0.43 ± 0.01	99.51 ± 0.05
simple macros	0.59 ± 0.02	99.94 ± 0.04
unnecessary_rules	0.65 ± 0.01	99.52 ± 0.08
risky_rules	1.06 ± 0.01	99.51 ± 0.05
parametrized macros	168.42 ± 2.17	446.52 ± 0.07

In order to evaluate the performance of SELint we conducted a set of measurements, collecting execution time and memory usage. We consider these numbers to be the most important indicators for SELint, since it can be used either manually by a single person or automatically as part of a Continuous Integration (CI) process. The measurements were conducted on an off-the-shelf laptop with an Intel Core i7-4770HQ 2.20GHz CPU and 16GB of 1600MHz DDR3 RAM. Each measurement was repeated 10 times, and the average and standard deviation are presented in Table 4 and Table 5. The first table presents data for a public Intel tree, Android 5.1[6], and the second one for the public AOSP tree, master branch[7]. For all measurements we have measured the SELint core and each of its plugins separately. The big difference in performance between these two trees comes from the number of expanded rules in the source policies: for the Intel tree it is 99532, while for the AOSP tree it is 3081233. The execution time of the `unnecessary_rules` plugin scales differently than others, requiring almost the same time as the SELint core on the Intel tree and 10 times more than the SELint core on the AOSP tree. This is due not only to the different total number of rules in the two trees, but also to the number of rules that each domain has, since the plugin needs to check for ineffective rule combinations or permissions (see Section 4.6). The `parametrized macro` plugin is the only plugin that takes a considerable amount of time to run, especially on the AOSP tree. As explained in Section 4.4, this is due to the fact that we are currently not implementing any heuristics in our solution to the problem, and are just relying on exploration of the solution space. As a result, the current plugin should not be included into the default set of plugins executing automatically as part of a CI process, but should be used manually by an expert. The execution time and memory usage of the other plugins fit the desired use cases: given that normally an AOSP build takes at least half an hour to complete in a powerful CI infrastructure, an overhead of minutes and hundreds of MB of memory is considered acceptable.

Table 5: Performance measurements for SELint on AOSP tree with 3081233 expanded rules.

Component	Avg time (s)	Avg mem (MB)
SELint core	1.88 ± 0.02	212.11 ± 0.07
user_neverallows	1.89 ± 0.02	212.09 ± 0.07
simple macros	2.18 ± 0.03	219.03 ± 0.09
unnecessary_rules	20.25 ± 0.17	212.07 ± 0.06
risky_rules	3.23 ± 0.03	212.07 ± 0.07
parametrized macros	3210.03 ± 48.13	6031.84 ± 0.59

[6] github.com/android-ia

[7] android.googlesource.com

6 DISCUSSION

While our evaluation showed that SELint is considered a valuable tool for analyzing SEAndroid policies, there are many areas for future work and improvements. The initial setup of SELint would benefit from an interactive procedure, allowing users to automatically detect and solve the possible mismatches between the installed libraries and policy versions. The `parametrized macro` plugin could provide an implementation based on a heuristic solution for the knapsack problem allowing users to obtain a partial solution, in order to save time and enable this plugin to be run as part of a CI infrastructure. More work is needed in order to polish the default configuration offered by the `risky_rules` plugin, and to provide a way for OEMs to easily, and maybe interactively, add scores for their own domains and types. We also need to conduct a study on how easy it is for SEAndroid experts to write new SELint plugins. Another future research direction is to investigate the possibility of using SELint together with a policy decompiler, in order to analyze OEM policies from available Android devices. This would provide additional input for SELint evaluation.

We continue to gather feedback from SELint users and SEAndroid experts to adjust SELint to their needs and requirements. Since SELint is open source software, and builds on existing official SEAndroid tools, we are planning to work with Google to include SELint in the set of SEAndroid tools provided with the AOSP tree.

REFERENCES

Amthor, P., Kuhnhauser, W., and Polck, A. (2011). Model-based safety analysis of SELinux security policies. In *NSS*, pages 208–215. IEEE.

Badger, L., Sterne, D., Sherman, D., Walker, K., , et al. (1995). Practical domain and type enforcement for UNIX. In *Security and Privacy*, pages 66–77. IEEE.

Bauer, M. (2006). Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, (148):13.

Chu, P. C. and Beasley, J. E. (1998). A genetic algorithm for the multidimensional knapsack problem. *J heuristics*, 4(1):63–86.

Clemente, P., Kaba, B., et al. (2012). Sptrack: Visual analysis of information flows within selinux policies and attack logs. In *AMT*, pages 596–605. Springer.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. (2005). Verifying information flow goals in security-enhanced Linux. *JCS*, 13(1):115–134.

Hanafi, S. and Freville, A. (1998). An efficient tabu search approach for the 0–1 multidimensional knapsack problem. *EJOR*, 106(2):659–675.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in Operating Systems. *CACM*, 19(8).

Hurd, J., Carlsson, M., Finne, S., Letner, B., Stanley, J., and White, P. (2009). Policy DSL: High-level Specifications of Information Flows for Security Policies.

Jaeger, T., Sailer, R., and Zhang, X. (2003). Analyzing integrity protection in the SELinux example policy. In *USENIX Security*, page 5.

Kellerer, H., Pferschy, U., and Pisinger, D. (2004). *Knapsack problems*. Springer, Berlin.

Magazine, M. J. and Chern, M.-S. (1984). A note on approximation schemes for multidimensional knapsack problems. *MOR*, 9(2):244–247.

Marouf, S. and Shehab, M. (2011). SEGrapher: Visualization-based SELinux policy analysis. In *SAFECONFIG*, pages 1–8. IEEE.

Reshetova, E., Bonazzi, F., Nyman, T., Borgaonkar, R., and Asokan, N. (2016). Characterizing SEAndroid Policies in the Wild. In *ICISSP*.

Schaufler, C. (2008). Smack in embedded computing. In *Ottawa Linux Symposium*.

SELinux (2014). Userspace tools. *github.com/ SELinux-Project/selinux*. Accessed: 29/09/15.

Smalley, S. and Craig, R. (2013). Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, volume 310, pages 20–38.

Smalley, S., Vance, C., and Salamon, W. (2001). Implementing SELinux as a Linux security module. *NAI Labs Report*, 1(43):139.

Sniffen, B. T., Harris, D. R., and Ramsdell, J. D. (2006). Guided policy generation for application authors. In *SELinux Symposium*.

Tresys (2016). SETools project page. *github.com/Tresys Technology/setools*. Accessed: 18/05/16.

Wang, R. (2016). Automatic Generation, Refinement and Analysis of Security Policies. *repository. lib.ncsu.edu/handle/1840.16/11139*.

Wang, R., Enck, W., Reeves, D., et al. (2015). EASE-Android: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *USENIX Security*.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Security and Privacy*, pages 95–109. IEEE.